Timed Automata Based Analysis of Embedded System Architectures

Martijn Hendriks [1] and Marcel Verhoef [1,2]

[1] Radboud University Nijmegen, ICIS
P.O. Box 9010, 6500 GL Nijmegen, NL
Martijn.Hendriks@cs.ru.nl

[2] Chess Information Technology B.V.
P.O. Box 5021, 2000 CA Haarlem, NL
Marcel.Verhoef@chess.nl

Abstract

We show that timed automata can be used to model and to analyze timeliness properties of embedded system architectures. Using a case study inspired by industrial practice, we present in detail how a suitable timed automata model is composed. Exact upper bounds on the timeliness properties can be found with the Uppaal model checker for a number of usage scenarios. We compare our results with three other performance modeling techniques. This comparison shows that if the state space of the model is tractable, Uppaal gives the most accurate results at similar cost. The proposed modeling strategy can be automated, which alleviates the difficulty and error-proneness of manually constructing timed automata models.

1 Introduction

In the area of embedded systems, the market pressure to reduce the cost price of, for example, consumer electronics is very high. This requires the available resources in the system to be used to its fullest. It is often not economically viable any longer to oversize the design to compensate for unknowns in current and future performance requirements. This problem is enhanced by the fact that the demand for functionality is always increasing and the time-to-market targets for the production of these products are constantly decreasing. Therefore, designers need to shift their attention from the informal question “Does the product work?” to the question “Does the product work, given a set of hard resource restrictions?”. This increases the importance of performance analysis in industry, in particular in the early phases of the system life cycle.

While in the past performance analysis was regarded as an activity that was primarily needed for the design of hard real-time systems, it is now recognized to apply to a much larger class of systems. Higher demands for usability aspects require the use of these techniques also in soft real-time systems, for example to analyze user interface responsiveness for gaming consoles or synchronization between audio and video stream decoding for DVD playback.

Our work is focused on performance evaluation of system designs in the early phases of the product life-cycle. This phase is characterized by its volatility, because there are still many unknowns that have great potential impact on the design. Paradoxically, it is also the life-cycle phase where the major architectural design decisions are taken. Obviously, performance analysis techniques are needed to support this decision making process. This support, however, can only be effective if it is very easy to construct, modify and analyze such a performance model. The turn-around time should be short to keep up with the interactivity of the design process.

In previous work [7], we explored the design of an in-car radio navigation system using real-time calculus. The aim of the experiment was to determine which proposed distributed embedded system architecture is best suited for the applications and their associated timeliness requirements. The results inspired us to explore other performance techniques using the same case study. In this paper we investigate timed automata.

Timed automata [1] are considered to be potentially useful for the analysis of systems, because of the expressiveness offered by this technique. Wide spread use in industry seems to be hampered by two issues: a) the general accepted belief that any realistically sized model will lead to a state space explosion during analysis and b) that despite its expressiveness, it remains hard to create timed automata models that reflect a particular design problem.

The AMETIST project has shown that significant progress has been made to tackle in particular the first question. Advances in tool development (such as symmetry reduction, efficient state representation and partial order evaluation among others) have enabled analysis of a number of very challenging industrial case studies, see *http://ametist.cs.utwente.nl*. For the second problem, two solution strategies are proposed in general. First, the timed automaton framework itself can be extended such that it provides concepts that are closer to the problem at hand. Second, another (more domain specific) language can be used as a front-end; a timed automata model should then be automatically derived from such a specification.

In this paper we show that the second solution is indeed possible. We derive timed automata models from UML sequence diagrams that are augmented with performance data. Together with the increased performance of the associated analysis tools, we believe that this improves the use of timed automata in practice.

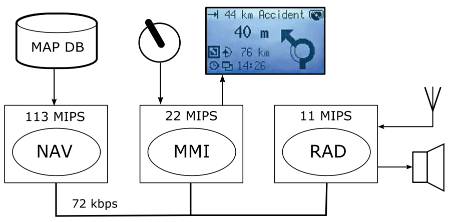


Figure 1. In-car radio navigation system

Related work. Petriu and Woodside [6] have shown that performance models can be generated from a set of scenarios and their use of resources. They take the UML profile for Schedulability, Performance and Time as their basis. Hence, we have not attempted to automate the model construction process ourselves because we expect that in due course commercial products will become available. Bennett, Field and Woodside [3] follow a similar approach to what we propose in this paper, but then using the stochastic process algebra FSP instead of timed automata (and LTSA instead of Uppaal respectively). They use a combination of model checking for verifying functional correctness properties and discrete event simulation for investigating the performance properties of the system, while we stay within one technique.

2 The case study

Figure 1 presents an overview of the in-car radio navigation system. The diagram suggests that there are three processors in the system interconnected by a communication bus. Each processor is assigned its own cluster of functionality, of which there are three: the man-machine interface (MMI), the navigation functionality (NAV) and the radio functionality (RAD).

We will consider three independent applications that run on the system concurrently; *ChangeVolume*, *AddressLookup* and *HandleTMC*. We will investigate whether the combination *ChangeVolume* and *HandleTMC* and the combination *AddressLookup* and *HandleTMC* will meet the system-level timeliness requirements for this architecture. First, we will characterize each application by presenting a UML sequence diagram that is augmented with performance data. In combination with the deployment and performance data shown in Figure 1, it provides sufficient information for modeling and analysis, which will be presented in Sections 3 and 4.

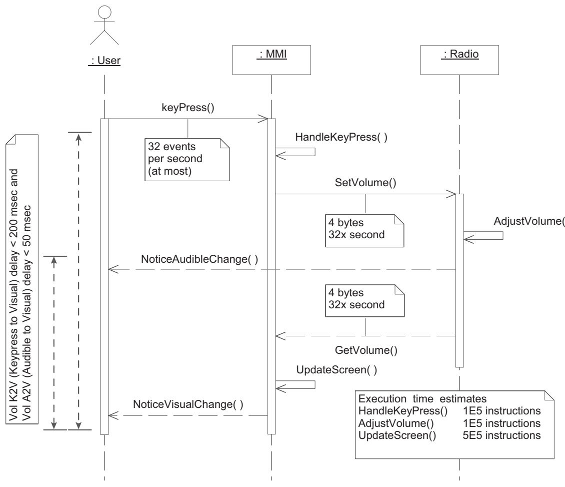

: User
: MMI
: Radio
keyPress()
32 events per second (at most)
HandleKeyPress( )
SetVolume()
4 bytes 32x second
AdjustVolume( )
NoticeAudibleChange( )
4 bytes 32x second
GetVolume()
UpdateScreen( )
NoticeVisualChange( )
Execution time estimates
HandleKeyPress() 1E5 instructions
AdjustVolume() 1E5 instructions
UpdateScreen() 5E5 instructions
Vol K2V (Keypress to Visual) delay < 200 msec and
Vol A2V (Audible to Visual) delay < 50 msec

Figure 2. The "ChangeVolume" scenario

Figure 2 represents the typical use of the volume control of the radio. Note that three operations are identified: *HandleKeyPress*, *AdjustVolume* and *UpdateScreen*. Worst-case execution times, event rates and message sizes are estimated and annotated in the sequence diagram, together with the principle timing requirements (*K2V* and *A2V*) applicable to this scenario.

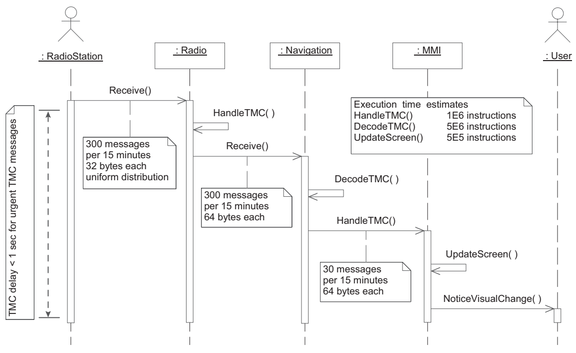

: RadioStation
: Radio
: Navigation
: MMI
: User
Receive()
HandleTMC( )
300 messages per 15 minutes 32 bytes each uniform distribution
Receive()
300 messages per 15 minutes 64 bytes each
DecodeTMC( )
HandleTMC()
UpdateScreen( )
30 messages per 15 minutes 64 bytes each
NoticeVisualChange( )
Execution time estimates
HandleTMC() 1E6 instructions
DecodeTMC() 5E6 instructions
UpdateScreen() 5E5 instructions
TMC delay < 1 sec for urgent TMC messages

Figure 3. The "HandleTMC" scenario

The TMC message handling scenario is shown in Figure 3. Note that the radio only receives the TMC messages; decoding is performed on the navigation subsystem because the map database is needed for that. Only fully decoded and relevant messages are presented to the user.

In this paper we analyze the combination of the

"ChangeVolume" and "HandleTMC" scenarios and the combination of the "AddressLookup" and "HandleTMC" scenarios. Since the modeling of the scenarios is very similar, we decided to omit the "AddressLookup" scenario here for brevity and only mention the analysis results of each combination. The interested reader can find a detailed informal description of all scenarios on-line at *http://www.mpa.ethz.ch*. Of course, in practice many more applications might be running in parallel to the three considered here. Note that the approach presented in the next section is not theoretically limited in the number of applications that can be analysed.

3 Modeling the system

The Uppaal model checker, which we use in this paper, is a tool for modeling and verifying networks of timed automata [2]. We assume that the reader has some knowledge about timed automata and their semantics. We will highlight where we use Uppaal specific language features. However, we do not describe the complete model[1], we merely present the patterns used for modeling.

First, we will provide an informal intuition to the modeling approach. We represent each hardware component as a separate timed automaton. The hardware component is either idle or busy computing some function. Similarly, each communication link is modeled as a timed automaton. Each link is either idle or busy transporting some data.

A system can then be represented by the network of timed automata composed of the automata representing the hardware and the automata representing the communication links. As a consequence, for each deployment diagram we need to construct a separate network of timed automata, because the functionality is distributed differently over the given hardware.

The external workload of the system is also characterized using timed automata; events are generated and inserted into the system according to the workload specification. The response of the system is observed by the same automaton such that the timeliness requirements can be expressed and verified by the model checker.

We will give an example of a hardware component in Section 3.1. Then, we will show how the communication is modeled in Section 3.2 and finally, we will present the automata representing the environment in which the system operates in Section 3.3.

3.1 Modeling the hardware

Figure 4 presents the basic automaton that models the behavior of the radio functionality (RAD). From the two sequence diagrams (Figures 2 and 3), it can be deduced that this functionality in fact consists of two operations, *AdjustVolume* and *HandleTMC* respectively. Each operation is represented as a location in the automaton. The automaton has a local clock *x* and two local constants, *HTMC* and *AV*. These constants represent the execution time of the operation, which is simply calculated as the worst-case execution time (expressed by the number of instructions to execute, as specified in the applicable sequence diagram) divided by the capacity of the hardware component (which is expressed in million instructions per second, as specified in Figure 1).

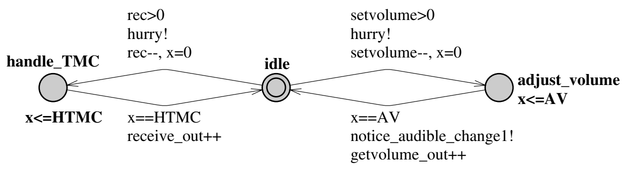


Figure 4. The automaton RAD

This calculation is obviously an approximation that in general does not hold. The peak capacity of modern CPU architectures can only be achieved for specific types of algorithms; digital signal processors are optimized for repeated multiply-addition operations that occur frequently in fast Fourier transformations for example. But despite this approximation, it suffices for our purpose since we are interested in high-level analysis of early design models. If we want to make a more accurate prediction, it is always possible to benchmark the operation on the target architecture and use the measured value instead.

The global variables *rec* and *setvolume* keep track of the number of pending calls to *HandleTMC* and *AdjustVolume* respectively. The transition is enabled if either counter is greater than zero. The *hurry!* synchronization is so-called urgent and always available. It thus enforces greedy behavior: transitions with this synchronization must be taken as soon as they are enabled. This ensures that the model does not delay when there still are outstanding requests.

Thus, if the RAD automaton is in location *idle* and the *Receive* event arrives (which is modeled by the increment of the *rec* variable), then it immediately takes the transition to the location *handle_TMC* ("immediately" means that no time elapses between the arrival of the event and the execution of the transition). The event is removed from the input queue (by decrementing the global variable *rec*) and the clock *x* is reset. The automaton stays for *HTMC* time units in location *handle_TMC* and then returns to the *idle* location while generating an output event (*receive_out* is incremented). As we will see in Section 3.2, the global variables *rec*, *setvolume*, *receive_out* and *getvolume_out* are the interface to the communication link. Similarly, we will see in Section 3.3 that the synchronization *no-*

[1]The Uppaal model and the informal case description is available on-line at *http://www.cs.ru.nl/ita/publications/papers/martijnh/*

tice_audible_change1! is used to emit the completion of the *AdjustVolume* operation back to the environment for measuring the response time.

Note that the RAD automaton in Figure 4 models a non-deterministic non-preemptive scheduler, which is in most cases not very realistic. The Uppaal language allows to model many kinds of schedulers. For instance, the automaton in Figure 5 models the radio functionality again, but now with a fixed priority preemptive scheduling strategy, in which the *AdjustVolume* operation has priority over the *HandleTMC* operation.

In Figure 5, modeling priorities is achieved by the additional guard expression *setvolume == 0* to the guard of the transition from location *idle* to location *handle_TMC*. This means that TMC messages may only be handled if there are no outstanding *ChangeVolume* requests pending. In order to model preemption, an additional local clock *y* and a local integer variable *D* are introduced to the automaton. The transition from *idle* to *handle_TMC* sets *D* to *HTMC*. Whenever a *AdjustVolume* event arrives during the execution of *HandleTMC*, the transition to *hdl_pre* is taken. Clock *y* is then used to measure the execution time of the *ChangeVolume* call that interrupted the execution of *HandleTMC*. When *ChangeVolume* is completed, the variable *D* is incremented with the delay caused by the preemption. Thus, preemption can easily be modeled, but it is crucial that it is known in advance how long a task can be preempted. Furthermore, it must not be the case that a task can infinitely often be preempted since then *D* grows to infinity and model checking is not possible anymore. The model checker can be used to prove that this is not the case by verifying some finite upper bound of *D* (care has to be taken because an integer variable in Uppaal has a finite domain by definition). The modeling of the other hardware components follow the same pattern as described above and are therefore not depicted here.

3.2 Modeling the communication

Modeling the communication links in the system is surprisingly similar to the models we have presented for the hardware components. For each link, we create a separate timed automaton. A location is created in the automaton for each message that is sent between the hardware components that communicate through the same communication link. The location reflects the fact that the message is being sent.

The location is occupied for as long as the message transfer takes. In Figure 6, we use the constants *BYTES4* and *BYTES64* to represent the time to transfer 4 and 64 bytes respectively over the communication link. This constant is again simply calculated as the message length (in bits, which is specified in the augmented sequence diagrams) divided by the baud rate (which is specified in the deployment

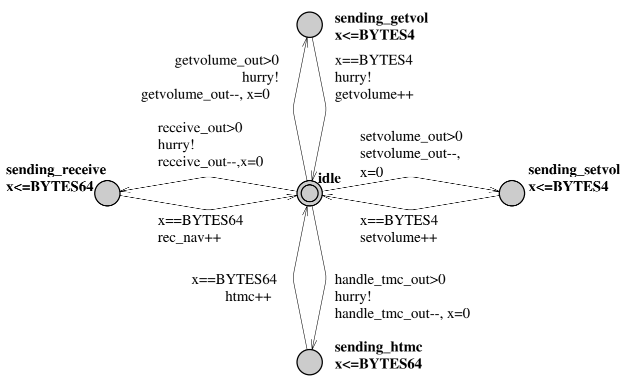


Figure 6. The automaton BUS

diagram). Obviously, this formula can be adapted to compensate for expected protocol overhead.

The hardware components interface to the communication link using (sharing) the global variables that count the number of outstanding messages of a particular type that need to be transfered over the link. If the *SetVolume* message from Figure 2 is to be sent from the MMI to RAD, then the MMI automaton will simply increment the global *setvolume_out* variable. If the bus is *idle* (and all other global variables are zero), then the transition towards the location *sending_setvol* location is immediately taken (again due to the *hurry!* synchronization); this will decrement the *setvolume_out* variable. The location *sending_setvol* is occupied for *BYTES4* time units and then the transition back to *idle* is taken; this will in turn increment the global variable *setvolume* which in turn will enable the corresponding transition in the RAD automaton, as presented in the previous paragraph.

Note that we can again use the same strategy for dealing with priorities. The communication link in Figure 6 is very simple, if more than one message is available then there will be non-deterministic choice who will get the turn to claim the bus. The bus is then blocked for as long as the message transfer is in progress. This resembles many of the simple industrial serial bus interfaces such as RS-485 for example. Analogous to Section 3.1, we can implement priorities by adding additional guards to the transitions that point outward from the *idle* location. It is relatively easy to mimic priority based protocols such as the Controller Area Network (CAN) or time-triggered protocols. For example a solution for a TDMA bus concept is proposed by Perathoner et al in [5]. Less trivial however is the encoding of protocols that break large messages into pieces to prevent starvation. However, the approach presented has another interesting characteristic. If the interface (the global variables) remains the same, then it would be simple to replace a certain bus concept by another by merely replacing the bus au-

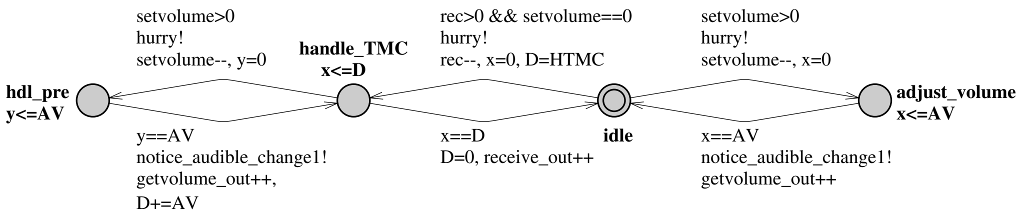


Figure 5. The automaton RAD-*preemptive*

tomata. It would not affect the hardware components at all. We can then easily investigate the impact of different bus protocols for a given deployment.

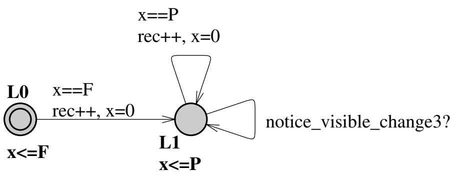
3.3 Modeling the environment

There are two actors that exercise the system from the environment: (i) a *user* who initiates the *ChangeVolume* scenario and (ii) a *radiostation* that initiates the *HandleTMC* scenario. There are two timed automata for each actor: a "normal" automaton and a "measuring" automaton. Which one is included in the system (the network of timed automata we analyze using Uppaal) depends on the worst case execution time we want to investigate. If we want to measure the response time of the *ChangeVolume* scenario, we add the "measuring" automaton for the *user* and the "normal" automaton for the *radiostation*. Vice versa, if we want to measure the response time of the *HandleTMC* scenario, then we add the "measuring" automaton for the *radiostation* and the "normal" automation for the *user*.

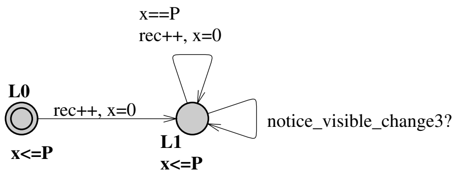
We consider four basic kinds of event arrival models: (i) periodic, (ii) sporadic, (iii) periodic with jitter and (iv) bursty event streams. For the strict periodic event model, an offset can be specified to force a phase shift in the signal (start of the first period). The periodic and sporadic event models can be expressed by automata as shown in Figure 7 (a-c). The event model for periodic behavior with jitter (where the jitter must be smaller than or equal to the period) can elegantly be expressed by the model proposed by Perathoner et al in [5], which is shown in Figure 7 (d).

The behavior of events is called bursty when the jitter becomes larger than the period of the event. This can be modeled, but it is much more involved than the previous model of periodic behavior with small jitter. The reason is that the subsequent intervals in which an event can occur now overlap. Figure 8 shows the Uppaal model for bursty behavior with period P, jitter J and a minimal seperation time between events of D.

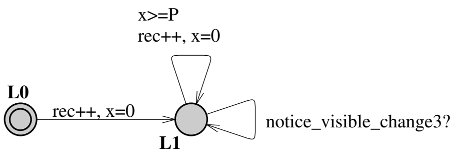
This automaton has two local variables, *pending* and *snd*. The variable *pending* is incremented every P time units (using clock x) which models that an event may be sent. The clock y is used to keep track of the next deadline for sending an event. If an event may be sent (i.e., $z > D$ which



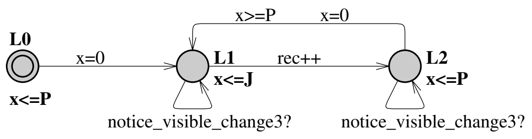
(a) Strictly periodic events with offset F



(b) Strictly periodic events with an undefined offset



(c) Sporadic events



(d) periodic with jitter where $J \leq P$

Figure 7. Example environment automata

models the inter-event seperation time and $pending > 0$), then the event is sent and *snd* is incremented. This signals that the deadline for the next event may be incremented by P time units. This is modeled by the reset of clock y after J (for the first event) or after P (for the other events) time units. Note that this automaton is not very nice for the state space of the model: it has three local clocks (but if $D = 0$ then z can be left out) and two local integer variables that both can count up to $\frac{J}{P} + 1$.

Every event automaton also has a "measuring" variant which is used to record the WCRT of a generated event.

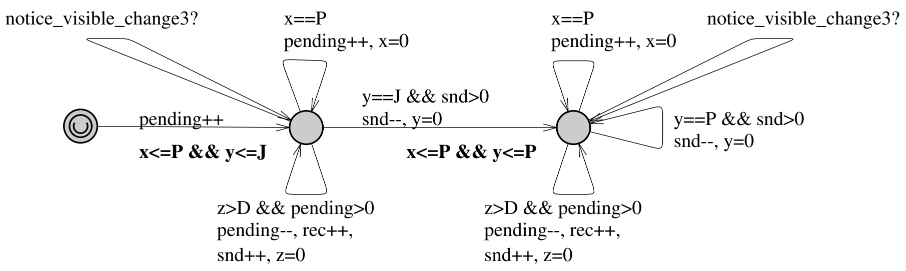


Figure 8. environment automaton describing event bursts

These automata seem complicated at first sight, but are in fact very logical in structure. All measuring variants can therefore be automatically generated.

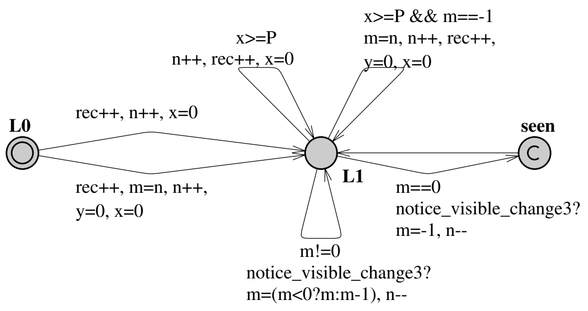


Figure 9. The measuring variant "rstat-m"

The automaton in Figure 9 is the measuring variant of the automaton of Figure 7(c). It has two additional integer variables, *m* (initially -1) and *n* (initially 0) and an additional clock variable *y*. This automaton can generate input events in the same way as the automaton in Figure 7(c) by using the upper transition from *L0* to *L1* and the upper-left and lower self-loop of *L1*. The variable n keeps track of the number of events that have been fed into the system and for which no response (a synchronization over the *notice_visible_change3* channel in this case) has been received yet. The automaton can also arbitrarily choose an event to measure the worst case response time (WCRT) by the lower transition from *L0* to *L1* and the upper-right self-loop of *L1*. (It is assumed that all queues are FIFO queues, i.e., events do not overtake each other, they are order preserving and that events are never dropped.) On those transitions y is reset to 0 and m is reset to the value of n. Hence, m counts the number of responses that need to be seen before the one is een that is used for the measurement. Note that at most one measurement can be in progress and that *m = -1* if no measurement is in progress. If *m == 0* then the response of the event that is being measured is expected. If the response occurs, then the transition to *seen* is taken. This is the so-called "committed" location in which no time may elapse. The WCRT of the *HandleTMC* scenario can thus be found using Uppaal by finding the smallest *C* (using a simple binary search strategy manually) such that Property 1 is satisfied.

$$\mathbf{AG}(\textit{rstat-m.seen} \longrightarrow \textit{rstat-m.y} < C) \qquad (1)$$

The modeling of the other actors follows the same pattern as explained above and are therefore not depicted here.

4 Analysis of the Uppaal model

The Uppaal model has been analyzed for 5 different requirements and 5 different sets of environment models. The results have been collected in Table 1. (Uppaal version *3.6-alpha-1* has been used with default options.) The rows indicate which requirement is measured and in what context (combination of applications) the analysis was performed.

The first column shows the results for strictly periodic event streams with a *user-defined* offset of 0 for all events (*po*, $F = 0$), which reflects fully dependent (synchronous) environment models. In the second column, the results are shown for strictly periodic event streams with an *unknown* offset for all events (*pno*), which reflects fully independent (asynchronous) environment models. The third column presents the analysis results for sporadic event streams (*sp*) where only a lower bound is specified for the event inter arrival time. In the fourth column we show periodic event streams with small jitter (*pj*, $J = P$) for the "radio station" environment model and sporadic events for the others. And finally, we use bursty event streams (*bur*, $J = 2P$, $D = 0$) for the "radio station" and sporadic event streams for the others in the last column.

Analysis of the *HandleTMC* and *AddressLookup* scenarios proved to be no problem. The verification times for *po*, *pno* and *sp* where so small (typically less than a second) that a binary search could easily be performed. The *pj* and *bur* scenarios took a bit more effort, but still a binary search was feasible (verification times typically in the order of a few minutes). The *HandleTMC* scenario in combination with *ChangeVolume* proved to be a problem for the *pj* and

Table 1. Uppaal worst-case response time analysis results (in milliseconds)

Event model / *Requirement*	*po* ($F = 0$)	*pno*	*sp*	*pj* ($J = P$)	*bur* ($J = 2P, D = 0$)
HandleTMC (+ ChangeVolume)	357.133	381.632	382.076	$>$ 400.000 (df)	$>$ 500.000 (rdf)
HandleTMC (+ AddressLookup)	172.106	239.080	239.080	329.989	420.898
K2A (ChangeVolume + HandleTMC)	27.716	27.716	27.716	$>$ 27.715 (bf)	$>$ 27.715 (bf)
A2V (ChangeVolume + HandleTMC)	41.796	41.796	41.796	$>$ 41.795 (bf)	$>$ 41.795 (bf)
AddressLookup (+ HandleTMC)	79.075	79.075	79.075	79.075	79.075

bur scenarios. This is due to the large difference in time scales of the event automata: the period of the "radiostation" events is in the order of seconds whereas the period of the "change volume" events is in the order of milliseconds. Such differences are bad for the symbolic representation of clock values that is used by Uppaal.

However, Uppaal can still be used as a "structured testing" tool with its options for the search order (*df = depth first*, *rdf = random depth first*). The verifier will try to find a *counterexample* of the property. If it finds one, then the constant C in the property is a lower bound on the WCRT of the event. This is indicated by the "greater than" symbols in Table 1.

Note that the *AddressLookup* and *ChangeVolume* worst-case response time values remain constant since (i) they have priority over the "radiostation" related events and (ii) their event model parameters are such that events are never queued for processing. For example, each *AddressLookup* event is fully processed before the next event arrives. If we would allow jitter to the *AddressLookup* scenario, such that two events might overlap, then the bound of 79.075 would increase.

Note that the results for *po* and *pno* are not identical, which indicates that a phase shift between the environment models *does* matter in this case. We get this result almost for free, neither the modeling nor the analysis effort is much influenced. The approach shown here can treat asynchronous and synchronous environment models by simply removing an invariant from the appropriate environment models.

5 Comparison with other techniques

In our earlier work [7], we have computed the worst-case response time results for several different distributed architectures using Modular Performance Analysis, but in this paper we only performed the analysis and comparison of the architecture shown in Figure 1. The main reason for this is that the timed automata models are constructed by hand which is a slow and very error prone process. We do believe that it is possible to automate the construction of these models, which is future work.

We did compare our analysis results with a few other techniques that were applied to the same case study. The results are summarised in Table 2. For space reasons it is not possible to discuss any of these models in detail. Note that the results for the other techniques where provided to us by other researchers that can be considered expert users for each respective tool.

The parallel object-oriented specification language (POOSL, *http://www.es.ele.tue.nl/poosl/*) and its SHESIM support tool was used to build a discrete event simulation model, which is typically used in industry nowadays. The simulation run of the periodic case with an unknown offset (*pno*) shows that the worst-case response time is not necessarily found by simulation, which is due to the fact that infinitely many values can be chosen for the offset parameter. Moreover, simulation runs typically took minutes to several hours.

Symbolic timing analysis for systems (SymTA/S, *http://www.symtavision.com*) is a performance and timing analysis tool based on formal scheduling analysis techniques and symbolic simulation. The tool supports heterogeneous architectures, complex task dependencies, context aware analysis, and combines optimization algorithms with system sensitivity analysis for rapid design space exploration [4]. The results found by SymTA/S were on par with the results found by Uppaal, with the exception of the periodic event streams with zero offset. In-depth discussions with the SymTA/S experts learned us that this type of event streams can in fact be handled but the specific technique needed is not yet fully implemented in SymTA/S. In stead, the *pno* results are used, which provides a safe upper bound for the WCRT.

Modular performance analysis is a deterministic queuing theory based on real-time calculus (MPA, *http://www.mpa.ethz.ch*). Again, the results found with MPA are quite similar to Uppaal and SymTA/S. The results are slightly more conservative mainly because context information is lost due to the transformation from the time to the time interval domain which is used by real-time calculus. The phase shift disappears when this transformation is performed. Therefore, it is not possible to get tight bounds for the *po* case with MPA, it will always yield the more conservative *pno* results.

Table 2. Worst-case response time results – comparison with other tools

Tool / Requirement	*Uppaal (po)*	*Uppaal (pno)*	*POOSL (pno)*	*SymTA/S (pno)*	*MPA (pno)*
HandleTMC (+ ChangeVolume)	357.133	381.632	266.94	382.086	390.0862
HandleTMC (+ AddressLookup)	172.106	239.080	244.26	253.304	265.8491
K2A (ChangeVolume + HandleTMC)	27.716	27.716	27.7067	27.717	28.1616
A2V (ChangeVolume + HandleTMC)	41.796	41.796	41.7771	41.798	42.2424
AddressLookup (+ HandleTMC)	79.075	79.075	78.8989	79.076	84.066

6 Conclusion

We have shown that timed automata can be used to analyze timeliness properties of embedded system architectures. The proposed modeling approach is straightforward and can be automated. The benefit is that if the state space explosion problem can be controlled then exact and hard upper bounds are found during analysis, while this cannot in general be guaranteed for other techniques. Table 2 shows that the results found by our approach are indeed competitative to others. Some specific conclusions can be drawn from our experiments:

Uppaal is a very generic tool. Many model types and scenarios can be expressed (i.e. event models, communication, computation, mix of preemptive and non-preemptive elements) and analysed. However, manual construction and maintenance of these models is error prone and should be automated to be useful in an industrial setting.

When the time scales of the environment models do not differ too much, then Uppaal is quite suitable for case-studies of this size. Of course, model checking still suffers from the state space explosion problem. Nevertheless it can also be used as a "massive simulation" tool: let the model checker search (e.g. random-depth-first) for any trace that gives a lower bound on the response time.

The comparison shows that the results fit the theoretical picture; the simulation-based approaches give values that are smaller than those computed by Uppaal (the worst-case instance is not necessarily reached) while SymTA/S and MPA provide values that are larger than those computed by Uppaal (hard but not necessarily tight values are computed).

Uppaal was able to find more accurate results with similar modeling and analysis effort for most scenarios of this case study. In fact, some results found by simulation could be falsified by showing the counter example from the model checker while searching for that particular outcome. Currently, however, Uppaal lacks the features that are necessary to conveniently perform a parameter sweep; something that MPA and SymTA/S are capable of.

Comparing the various tools has not been easy. The key difficulty has been to ensure that all models have the same semantics and deal with modeling assumptions in a similar way. Using the Uppaal verification engine, these assumptions could be checked mechanically which helped to improve all specifications.

Acknowledgments. The authors would like to thank Ernesto Wandeler, Simon Perathoner, Kai Richter, Menno de Hoon, Frits Vaandrager, Jozef Hooman and the anonymous reviewers for their valuable comments to this paper. Verhoef works for the BODERC project under the responsibility of the Embedded Systems Institute. This project is partly supported by the Netherlands Ministry of Economic Affairs under the Senter TS program. Hendriks has been supported by the EC project IST-2001-35304 AMETIST, *http://ametist.cs.utwente.nl* .

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[3] A. J. Bennet, A. J. Field, and M. C. Woodside. Experimental Evaluation of the UML Profile for Schedulability, Performance and Time. In *≪UML≫ 2004 – The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2004.

[4] A. Hamann, R. Henia, R. Racu, M. Jersak, K. Richter, and R. Ernst. SymTA/S - Symbolic Timing Analysis for Systems. In *WIP Proc. Euromicro Conference on Real-Time Systems 2004 (ECRTS '04)*, 2004.

[5] S. Perathoner, E. Wandeler, and L. Thiele. Timed automata templates for distributed embedded system architectures. Technical Report 233, ETH Zurich, November 2005.

[6] D. B. Petriu and M. C. Woodside. A Metamodel for Generating Performance Models from UML Designs. In *≪UML≫ 2004 – The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2004.

[7] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis - A Case Study. In *International Symposium on Leveraging Applications of Formal Methods – ISOLA 2004*, pages 209–220. Department of Computer Science - University of Cyprus, 2004.